GLOBAL JOURNAL OF ENGINEERING SCIENCE AND RESEARCHES

DATA MOVEMENT IN WOS - ROS ON SQL OPERATIONS IN DATA WAREHOUSING DATABASE-VERTICA

Jisha Mariam Jose
Assistant Professor, Department Of Computer Science, New Horizon College Of Engineering, Bangalore, Karnataka, India

ABSTRACT

Vertica, an HP enterprise product is a data warehousing database that is used for big data analysis also. Vertica has many additional features, which makes it different from traditional database management systems. While the system's overall features have been described elsewhere in more breadth[1],[2]. The main aim of this paper is to describe various SQL(structured query language) operations which can be performed in Vertica database that supports its features like hybrid storage, columnar orientation etc.

Keywords: *Copy Trickle, Copy Direct, Insert Direct, Merge, Purge, Replay Delete, Ancient History Mark.*

I. INTRODUCTION

Vertica is a columnar oriented SQL RDBMS(Relational Database Management System) designed for analytics. Generally, for analytic workloads operations are majorly performed over many rows as per given requests, rather than workloads consisting of high numbers of requests per second, each retrieving or altering only a handful of rows[3]. Vertica's original design was inspired by the C-Store[2] academic project.

Being columnar oriented database, Vertica stores each column data file in sorted and compressed manner. On querying, the appropriate column values corresponding to each row is obtained properly because here data is stored first[4]. The nth record can be constructed by retrieving the nth value from each column file. And the retrieval of the nth value from a column is done with the help of a structure called the position index, which is a sparse mapping of positions to file offsets[3].

Vertica is designed to run on a cluster of commodity machines. In this paper, the execution of SQL operations is shown on a three-node cluster in Vertica. Here, the loading of data with help of INSERT and COPY command is shown to discuss how these data get stored on the execution of them. Also, detail explanation with the example is shown for MERGE and UPDATE operations. At last, the deletion of these data is pictorially explained to make one understand how Vertica's DELETE operation is different from other traditional RDBMS.

II. LOADING OF DATA

Insert
At a time, only one tuple can be inserted into the Vertica database tables. As we insert a tuple to a particular table, the corresponding values are also loaded to all the projections of that table. By default, inserted tuples are first loaded to the WOS(Write Optimized Store)[5]. When the WOS is full, the inserted tuple overflows to the ROS(Read Optimized Store). If for the first time, no projections are associated with the target table, Vertica creates a default super projection where it inserts the new values.
Syntax:
INSERT [/*+ *direct* */]
INTO [[*db-name*.]*schema*.]*table-name*
[(*column*[,...])] VALUES ({ *expression1*, ...})

Example 1:

A table **"staff"** is created in schema **"nh001"** using simple INSERT command as follows. First, these data are inserted to WOS and then Tuple Mover moveout task will move these data to ROS. In this example, moveout operation is done manually to show data moving from WOS to ROS[5], [6].

```
dbadmin=> create schema nh001;
dbadmin=> create table nh001.staff(sno int primary key, sname varchar(10), age int);
CREATE TABLE

dbadmin=> insert into nh001.staff values(111,'hari',25);
dbadmin=> insert into nh001.staff values(112,'ishaan',28);
dbadmin=> insert into nh001.staff values(113,'aditya',25);
dbadmin=> insert into nh001.staff values(114,'aman',26);
```

Displaying the "staff" table

```
dbadmin=> select * from nh001.staff;
 sno | sname  | age
-----+----------+-----
 112 | ishaan |  28
 113 | aditya |  25
 114 | aman   |  26
 111 | hari     |  25
```

Displaying system table "column_storage" to note down rows count in WOS before tuple mover moveout task

```
dbadmin=> select node_name,projection_name,column_name,compressions,wos_row_count,ros_row_count from column_storage where anchor_table_schema='nh001' and anchor_table_name='staff' and projection_name='staff_b0';
```

Column_storage system table

```
 node_name       | projection_name | column_name | compressions | wos_row_count | ros_row_count
-----------------------+----------------------+-------------------+-----------------+--------------------+------------------
v_nhdb_node0002 | staff_b0           | sno              |                  |        1         |        0
 v_nhdb_node0002 | staff_b0          | sname            |                  |        1         |        0
 v_nhdb_node0002 | staff_b0          | age              |                  |        1         |        0
 v_nhdb_node0002 | staff_b0          | epoch            |                  |        1         |        0
 v_nhdb_node0003 | staff_b0          | sno              |                  |        3         |        0
 v_nhdb_node0003 | staff_b0          | sname            |                  |        3         |        0
 v_nhdb_node0003 | staff_b0          | age              |                  |        3         |        0
 v_nhdb_node0003 | staff_b0          | epoch            |                  |        3         |        0
```

Here, all the data are inserted into WOS and the "wos_row_count" can been seen node wise. "Epoch" is the column which stores the epoch(time) at which that column value was inserted to WOS. Now when Tuple Mover's moveout task is done manually by using following command, the data are moved from WOS to ROS, which is shown below.

Command to do MOVEOUT task manually

```
dbadmin=> SELECT DO_TM_TASK('MOVEOUT');
```

Displaying system table "column_storage" to note down rows count in ROS after tuple mover moveout task

```
dbadmin=> select node_name, projection_name, column_name, compressions, wos_row_count, ros_row_count from column_storage where anchor_table_schema='nh001' and anchor_table_name='staff' and projection_name='staff_b0';
```

Column_storage system table

```
node_name    | projection_name | column_name | compressions | wos_row_count | ros_row_count
------------------------+-----------------+------------------+-----------------+---------------------+--------------------
 v_nhdb_node0003 | staff_b0      | sno          | int delta     |              0 |       3
 v_nhdb_node0003 | staff_b0      | sname        | lzo           |              0 |       3
 v_nhdb_node0003 | staff_b0      | age          | int delta     |              0 |       3
 v_nhdb_node0003 | staff_b0      | epoch        | none          |              0 |       3
 v_nhdb_node0002 | staff_b0      | sno          | int delta     |              0 |       1
 v_nhdb_node0002 | staff_b0      | sname        | lzo           |              0 |       1
 v_nhdb_node0002 | staff_b0      | age          | int delta     |              0 |       1
 v_nhdb_node0002 | staff_b0      | epoch        | none          |              0 |       1
```

Example 2:
A table **"dept"** is created in schema **"nh001"** using INSERT /*+direct*/ command as follows. In this case, data are directly inserted into ROS.

```
dbadmin=> create table nh001.dept(dno int primary key, dname varchar(10), block char);
CREATE TABLE

dbadmin=> insert /*+direct*/ into nh001.dept values(10,'HR','A');
dbadmin=> insert /*+direct*/ into nh001.dept values(20,'Sales','A');
dbadmin=> insert /*+direct*/ into nh001.dept values(30,'Testing','B');
dbadmin=> insert /*+direct*/ into nh001.dept values(40,'Accounts','C');
```

Displaying the "dept" table

```
dbadmin=> select * from nh001.dept;
 dno |  dname    | block
-----+-------------+-------
  20 | Sales      | A
  10 | HR         | A
  30 | Testing    | B
  40 | Accounts   | C
```

Displaying system table "column_storage" to note down row count in ROS due to direct INSERT command

```
dbadmin=> select node_name,projection_name,column_name,compressions,wos_row_count,ros_row_count from column_storage where anchor_table_schema='nh001' and anchor_table_name='dept' and projection_name='dept_b0';
```

Column_storage system table

```
node_name    | projection_name | column_name |compressions| wos_row_count | ros_row_count
-----------------+---------------------+-------------------+----------------+--------------------+--------------------
 v_nhdb_node0001 | dept_b0      | dno          | int delta    |              0 |       2
 v_nhdb_node0001 | dept_b0      | dname        | lzo          |              0 |       2
 v_nhdb_node0001 | dept_b0      | block        | lzo          |              0 |       2
 v_nhdb_node0001 | dept_b0      | epoch        | none         |              0 |       2
 v_nhdb_node0002 | dept_b0      | dno          | int delta    |              0 |       1
 v_nhdb_node0002 | dept_b0      | dname        | lzo          |              0 |       1
 v_nhdb_node0002 | dept_b0      | block        | lzo          |              0 |       1
 v_nhdb_node0002 | dept_b0      | epoch        | none         |              0 |       1
 v_nhdb_node0003 | dept_b0      | dno          | int delta    |              0 |       1
 v_nhdb_node0003 | dept_b0      | dname        | lzo          |              0 |       1
 v_nhdb_node0003 | dept_b0      | block        | lzo          |              0 |       1
 v_nhdb_node0003 | dept_b0      | epoch        | none         |              0 |       1
```

Here, all the data are directly stored to ROS container without going through WOS.

Copy:

The COPY command is designed for the bulk loading of data from a file on a cluster host into Vertica database. It reads data from a delimited text file and inserts tuples either into the WOS (memory) or directly into the ROS (disk)[5],[7]. The COPY command automatically commits itself. To prevent this we can use COPY NO COMMIT. Vertica recommends to COMMIT or ROLLBACK the current transaction before using COPY. There are different methods of COPY statements based on the type of data load.

- COPY AUTO
- COPY DIRECT
- COPY TRICKLE

Copy Auto:

COPY uses the AUTO method to load data into WOS. This is the default AUTO load method for smaller the bulk loads. The AUTO option is most useful when you cannot determine the size of the file. Once the WOS is full, COPY continues loading directly to ROS containers on disk. ROS data are sorted and encoded. By default, WOS capacity is 25% or 2GB of RAM(whichever is less)[2],[3].

Copy Direct:

The data are directly loaded into ROS containers using COPY DIRECT method. When files are loaded in bulk this method is used to improve the performance of large files. This is done by avoiding the WOS and loading data into ROS containers directly so that automatic spillover from WOS doesn't happen[2]. But this type of load may lead to many smaller data sets resulting in many ROS containers, which have to be combined later. That means this method can lead to ROS pushback.

Copy Trickle:

COPY uses the TRICKLE method to load data directly into WOS. After completion of initial bulk load, TRICKLE load method is used further to load data incrementally. In this case, an error occurs when the WOS becomes full and the entire data load is rolled back. So this method is used only when having a finely tuned load and moveout process, so that the WOS can hold the data that is being loaded incrementally. When data is loaded into partitioned tables, this method is more efficient than COPY AUTO [5].

ROS Push Back

The number of possible ROS storage containers per projection is limited. A maximum of 1024 containers per projection is only allowed in Vertica database. If the demand for creation of more containers arrives, then the ROS will push back the data. The count can be monitored using "ros_count" column in "projection_storage" system table. The ROS Pushback can be minimized by

- adjusting the count of partitions for a projection to be less than 700
- Stop loading(also speeds up recover)
- Increasing Tuple Mover merge out threads

Syntax:

```
COPY [target_schema.]target_table
   FROM {'file path'}
    [ DELIMITER 'char' ]
    [AUTO | DIRECT | TRICKLE]
    [NO COMMIT]
```

Two log files are created on each node:

- Exceptions log
- Rejected data log

The exception will show the reason for the error and in which line the error has occurred. Rejection data log contains the data which has been rejected. Log files are created on each node for rejected data. Logs are found in <catalog directory>/CopyError Logs. Rejection logs can be copied to a table by using following command syntax:

ISSN 2348 – 8034
Impact Factor- 5.070

```
        COPY … REJECTED DATA AS TABLE  <rejected_table_name>;
```
Then this rejection information can be queried from the table using querying command:
```
        Select * from <rejected_table_name>
```

Example:
A table **"ex"** is created in schema **"test"** as follows.
```
dbadmin=> create schema hp7;
CREATE SCHEMA
dbadmin=> create table test.ex(fname varchar(10), lname varchar(10), enrollyear int, gradyear int, place varchar(10));
CREATE TABLE
```

The file to be uploaded to table "ex" is as shown below:
/home/sample.csv
```
Smith, John, 1977, 1981, Bowers Hall
Brown, Tim, 1978, 1982, Blue Hall
Howe, Mary, 1976, 1980, White Hall
Kelly, Sarah, 1977, 1981, Blue Hall
Jones, William, 1977, 1981, Johnson Hall
Brady, Mark, 1976, 1980, Madison Hall
Black, Howard, 1975, 1979, Bowers Hall
King, Martha, 1976, 1980, White Hall
McCoy, Keith, 1978, 1982, Madison Hall
McDonald, Susan, 1977, 1981, Johnson Hall
Williams, Joe, 1975, 1979, Bowers Hall
Johnson, Julie, 1978, 1982, White Hall
Barry, Polly, 1976, 1980, Blue Hall
Wilson, Martin, 1975, 1979, Johnson Hall
Adams, Harry, 1976, 1980, White Hall
Ball, Terry, 1977, 1981, Bowers Hall
Chan, Ty, 1978, 1982, Johnson Hall
David, Donald, 1975, 1979, Madison Hall
Emory, Fletcher, 1976, as1980, White Hall
Flaherty, Finbarr, 1977 1981, Blue Hall
```

Copying the file at one stretch into "ex" table
```
dbadmin=> copy test.ex from '/home/sample.csv' delimiter ',' ;
Rows Loaded
-------------
        18
```
Here, out of 20 rows, only 18 rows are loaded and other two rows are rejected and stored in logs.

The content of exception log file present in CopyErrorLogs folder is:
```
COPY: Input record 19 has been rejected (Invalid integer format ' as1980' for column 4 (gradyear)).
COPY: Input record 20 has been rejected (Invalid integer format ' 1977 1981' for column 3 (enrollyear)).
COPY: Loaded 18 rows, rejected 2 rows.
```

The content of rejected log file("ex-testsample.csv-copy-from-rejected-data") present in CopyErrorLogs folder is the rows that got rejected while copying data.
```
Emory, Fletcher, 1976, as1980, White Hall
Flaherty, Finbarr, 1977 1981, Blue Hall
```

III. MERGE AND UPDATE OPERATION ON SAMPLE DATA

Merge

The MERGE statement performs INSERT and UPDATE operations based on the given condition as a single operation. In other words, Vertica on the execution of MERGE statement overlooks the condition mentioned by the user; if the condition matches then it updates the action mentioned below it and when the condition doesn't match it inserts rows in the target table from rows in the source table. By default, the MERGE operation uses Write Optimized Storage(WOS). If WOS fills up, data overflows to the ROS[5].

Syntax:

```
MERGE  INTO [[db-name.]schema.]target-table
            USING [[db-name.]schema.]source-table
             ON ( condition )
           [ WHEN MATCHED THEN UPDATE SET
               column1 = value1 [, column2 = value2 ... ] ]
            [ WHEN NOT MATCHED THEN INSERT
                ( column1 [, column2 ...]) VALUES ( value1 [, value2 ... ] ) ]
```

Example 1:

Two tables "student_source" and "student_target" are created in schema "nh01" to show the MERGE operation.

```
dbadmin=> create table nh001.student_source(studid int primary key, mark int, age int, address varchar(10));
dbadmin=> create table nh001.student_target(studid int primary key, mark int,age int,address varchar(10));
```

Insertion of three rows to "student_source' table:

```
dbadmin=> insert into nh001.student_source values(101,80,23,'thirupathi');
dbadmin=> insert into nh001.student_source values(103,60,25,'vijaywada');
dbadmin=> insert into nh001.student_source values(105,70,23,'chennai');
```

Insertion of three rows to "student_target' table:

```
dbadmin=> insert into nh001.student_target values(101,75,22,'Bangalore');
dbadmin=> insert into nh001.student_target values(102,50,23,'chennai');
dbadmin=> insert into nh001.student_target values(103,60,25,'hyderabad');
dbadmin=> insert into nh001.student_target values(104,85,22,'bangalore');
```

Display of "student_target table"

```
dbadmin=> select * from nh001.student_target;
 studid | mark | age |  address
--------+-----+-----+-----------
    102 |   50 |  23 | chennai
    104 |   85 |  22 | bangalore
    103 |   60 |  25 | hyderabad
    101 |   75 |  22 | Bangalore
```

Display of "student_source table"

```
dbadmin=> select * from nh001.student_source;

 studid | mark|age| address
--------+-------+-----+------------
    103 |     60 |  25 | vijaywada
    105 |     70 |  23 | chennai
    101 |     80 |  23 | thirupathi
```

Merging of above two tables, if following condition is satisfied:-

If student id of source table matches with target table then target table is updated with values corresponding to source table. When there is no match, then the rows corresponding to new student ids from source table are inserted to target table also.

```
dbadmin=> MERGE into nh001.student_target T using nh001.student_source S on S.studid=T.studid WHEN MATCHED THEN UPDATE SET mark=S.mark, age=S.age, address=S.address WHEN NOT MATCHED THEN INSERT VALUES(S.studid, S.mark, S.age, S.address);
```

Display of "student_target" table after MERGE operation

```
dbadmin=> select * from nh001.student_target;
 studid| mark | age |  address
--------+--------+-----+------------
   101 |     80 |  23   | thirupathi
   102 |     50 |  23   | chennai
   104 |     85 |  22   | bangalore
   105 |     70 |  23   | chennai
   103 |     60 |  25   | vijaywada
```

Here, student id 101, 103 of source table matches with target table and so target table is updated for these two rows accordingly along with retaining other student id rows as it is in target table. Also a new row with student id 105 is inserted to target table.

Example 2:

Two tables "product" and "product_count" are created in schema "nh001" as follows:

```
dbadmin=> create table nh001.product(custid int primary key, product_name varchar(15), purchase_date date);
dbadmin=> create table nh001.product_count(custid int primary key, product_name varchar(15), count_purchase int);
```

Insertion of four rows to "product_count" table which keeps track of no:of product purchased by a customer.

```
dbadmin=> insert into nh001.product_count values(1111,'samsung note3',1);
dbadmin=> insert into nh001.product_count values(1112,'LG Led TV',1);
dbadmin=> insert into nh001.product_count values(1113,'Philips Bulb',10);
dbadmin=> insert into nh001.product_count values(1114,'iphone 6S',2);
```

Insertion of four rows to "product" table which stores details of purchase made by customer after the last update on "product_count" table.

insertion of four rows to "product" table

```
dbadmin=> insert into nh001.product values(1112,'LG Led TV','2018-06-01');;
dbadmin=> insert into nh001.product values(1114,'Philips Bulb','2018-05-22');
dbadmin=> insert into nh001.product values(1111,'samsung note3','2018-04-15');
dbadmin=> insert into nh001.product values(1111,'Philips Bulb','2018-03-10');
```

Display of "product" table

```
dbadmin=> select * from nh001.product;

 custid | product_name  | purchase_date
--------+---------------------+--------------------
  1114 | Philips Bulb      | 2018-05-22
  1112 | LG Led TV        | 2018-06-01
  1111 | samsung note3  | 2018-04-15
  1111 | Philips Bulb      | 2018-03-10
```

Display of "product_count" table

```
dbadmin=> select * from nh001.product_count;
 custid | product_name | count_purchase
--------+--------------------+---------------------
   1114 | iphone 6S         |            2
   1113 | Philips Bulb      |           10
   1111 | samsung note3 |            1
   1112 | LG Led TV        |            1
```

Merge of above tables if following condtion is satisfied:-

If customer id of product table matches with customer id of product_count table then purchase_count field of product_count table is incremented when that particular customer purchases the same product again. When not matched, the new customer or old customer with count of new product purchase is added to "product_count" table.

```
dbadmin=> merge into nh001.product_count c using nh001.product p on (c.custid=p.custid and c.product_name=p.product_name) when matched then update set count_purchase=count_purchase+1 when not matched then insert(custid,product_name,count_purchase) values (p.custid,p.product_name,1);
```

Display of "product_count" table after MERGE

```
dbadmin=> select * from nh001.product_count;
 custid | product_name  | count_purchase
--------+--------------------+----------------
   1111 | samsung note3 |            2
   1112 | LG Led TV        |            2
   1111 | Philips Bulb      |            1
   1114 | iphone 6S         |            2
   1114 | Philips Bulb      |            1
   1113 | Philips Bulb      |           10
```

Update

DELETE and UPDATE operations in HP Vertica differs from traditional databases in two ways:

- DELETE does not delete data immediately from the table as it is executed but marks the rows to be deleted.
- UPDATE operation indirectly performs two actions: First DELETE the records with old values and then INSERT new rows with given updated values. For this, it creates two new rows: one with new data and one marked for deletion.

On UPDATE, insertion of new records is performed into the WOS. If the WOS fills up, the operation overflows to the ROS. Until the Tuple Mover's purge operation is performed, the deleted rows remain in physical storage. The deleted data are permanently removed after purge operation to free up the disk space. On an UPDATE operation, the condition mentioned in the where clause is verified. If it is true, then the values of the specified columns in all rows for which a specific condition is true is replaced. All other columns and rows in the table are unchanged. On success, UPDATE operation returns the count of rows that got updated. A count of 0 indicates query is right but since no rows matched the condition, none of the rows got updated. UPDATE supports sub-queries and joins, which is useful for updating values in one table based on values that are stored in other tables[10].

Syntax:

```
 UPDATE [ /*+ direct */ ]
     [[db-name.]schema.]Table-Reference
        SET column = ... { expression}
          WHERE Clause
```

ISSN 2348 – 8034
Impact Factor- 5.070

Example:
A table **"dept"** is created in schema **"nh001"** as follows:

```
dbadmin=> select * from nh001.dept;
 dno |  dname   | block
-----+----------+-------
  20 | Sales    | A
  10 | HR       | A
  30 | Testing  | B
  40 | Accounts | C
```

Update statements:

```
dbadmin=>UPDATE dept SET block='D' where dno=10;
dbadmin=>UPDATE dept SET dname='Marketing' where dno=20;
```

Display of table after update operation

```
dbadmin=> select * from nh001.dept;
 dno |  dname    | block
-----+-----------+-------
  20 | Marketing | A
  10 | HR        | D
  30 | Testing   | B
  40 | Accounts  | C
```

IV. LIFE CYCLE OF DELETION OF DATA FROM PROJECTIONS

When DELETE statement is executed in Vertica, the data is not removed immediately from storage containers. Instead, the DELETE statement add a new file called delete vector that points to each WOS and ROS container that contains records marked for deletion[5]. Each delete vector contains two values: one is the position of a deleted record in a container and the second is the epoch where the DELETE statement was committed. When you run a SELECT query on a table, Vertica displays the result after filtering out the records that are marked in delete vectors.

Syntax

DELETE [/*+ direct */] FROM [[*db-name*.]*schema*.]*table*
WHERE Clause

Reasons for Deleting Data

Common reasons for deleting data are as follows:

- You need to delete historical data at regular intervals.
- You need to update and/or delete data loaded by mistake.
- You need to delete staging tables.

Table 1. Different Types of Delete

	Recommended Load	**Rollback Possibility**	**Performance**	**Use**
Single Row Delete	WOS	Yes	Depends on projection design	Recommended in WOS so that deleted rows are combined in one delete vector when data is moved out.
Trickle Load	WOS	Yes	Depends on Projection Design	Recommended for small batches that happen at frequent intervals. Recommended in WOS so that deleted rows are combined in one delete vector when data is moved out.
Bulk Delete	Direct ROS	Yes	Depends on projection design	Recommended because it creates one delete vector for each ROS that has data to be marked

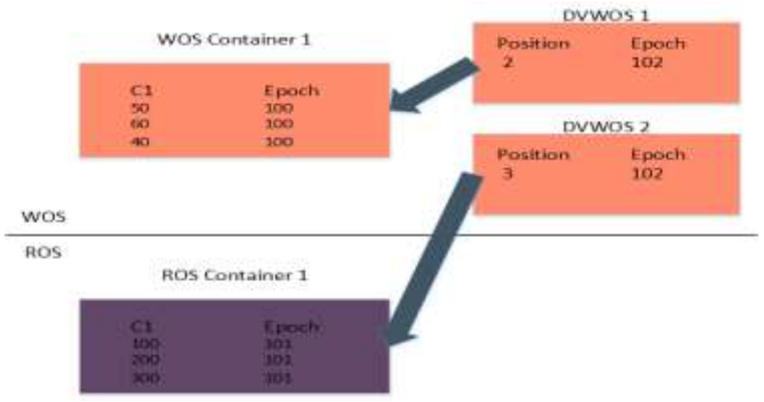
ISSN 2348 – 8034
Impact Factor- 5.070

				as deleted.
Drop Partition	N/A	No	Fast with catalog removal and storage is removed in the background.	Recommended to clean historic data. It forces the moveout operation before executing to move data inserted in ROS that belongs to partition to be removed.
Truncate	N/A	No	Fast with catalog object changes and storage is removed in the background.	Removes all storage associated with a table while preserving the table definition. Recommended when you need to clean the table content.

AHM epoch
The *ancient history mark (AHM)* is the epoch prior to which deleted data can be purged from physical storage. By default, Vertica advances the AHM epoch at an interval of 180 seconds. The Ancient History Mark (AHM) decides when the data is removed permanently. The AHM is an epoch that represents the time until which the history is retained[1][5].

Delete life cycle process
The following figures and steps show an example of the delete life cycle process. This example shows a table with one WOS container and one ROS container.
Step 1: Run the following DELETE statement:
=>DELETE from Table1 where C1 in (60,300);
This statement creates a delete vector in WOS (DVWOS) for the WOS container and ROS container. Vertica commits this statement in epoch 102 [8]:

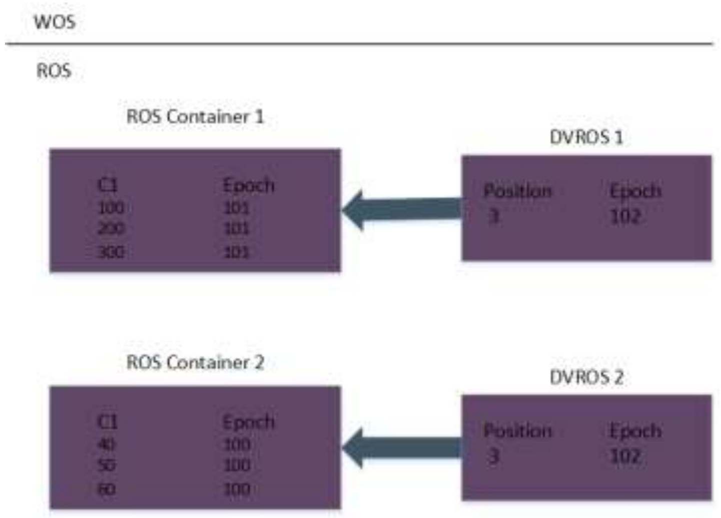
Figure 1



Creation of DVWOS at epoch no 102

Step 2:
The Tuple Mover responds with a moveout operation that moves data from WOS into ROS and creates delete vectors in ROS (DVROS). As the figure shows, the position of the data changes after the moveout operation(in ROS data are sorted and stored) and DVROS captures new positions:

Figure. 2

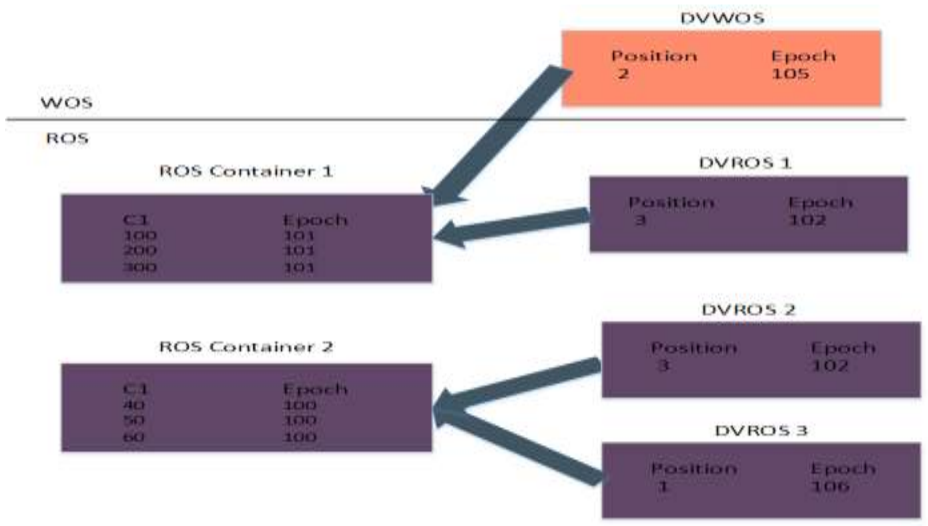


Creation of DVROS on Tuple Mover moveout task

Step 3: The AHM advances to 103. Run the following delete statements:
DELETE from Table1 where C1=200; commit;
DELETE /*+direct*/ from Table1 where c1=40; commit;

These statements are committed in epoch 105 and epoch 106, respectively. The DELETE statement creates a new DVWOS container, while the DELETE statement that uses direct hint creates a DVROS container.

Figure 3

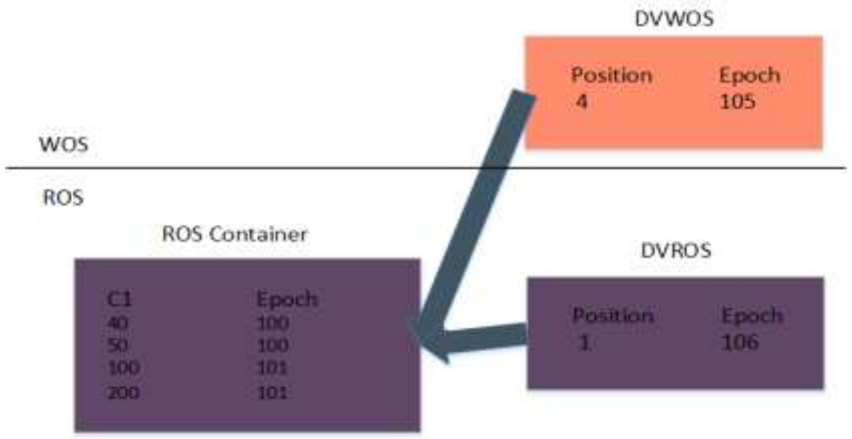


Creation DVWOS due to DELETE and DIRECT DELETE operations

Step 4:
The Tuple Mover responds with a mergeout operation that merges the ROS containers into a single, sorted ROS container. The mergeout operation purges any records deleted prior to the AHM(Ancient History Mark). The

following figure shows that the records deleted in epoch 105 and 106 could not be purged. The purge did not occur because the commit epoch of the DELETE statements is greater than the AHM epoch.

Figure.4



Removal of data permanently on PURGE operation

Pros And Cons of DELETE Operation with Delete Vector Mechanism
Pros
Since, the rows are marked as deleted and not removed from the storage on DELETE operation, the data remain available to historical queries. DELETE process can be long and cost full, so saving of time and resources at the time we chose the deleted data is important.
Cons
Vertica uses disk space for the deleted rows and delete markers(as Vertica will write a new file that will point out to the deleted data). Also , there is performance penalty when reading and skipping over deleted data has to be done during query execution.

Perform Bulk Delete
Bulk delete can be used instead of multiple single deletes also. In this case, one should commit in WOS, while deleting multiple single rows of data. Failing to commit creates multiple delete vectors for each statement. Best practices for performing a bulk delete are as follows:

- Load the delete predicate in a temporary table.
- Delete rows in one statement by joining the temporary table with the delete predicates and the table with data to be deleted.

For example: To delete records (Employee ID) for several employees there can be two ways:

- Delete the records using single delete statements, which creates one delete vector per statement.
- Create one bulk delete statement that creates one delete vector per ROS container that contains deleted data.

Example:
The following code snippet illustrates how to perform a bulk delete.
A table "shop" is created in schema "nh001" to store details of employees working in a shop as follows:
```
dbadmin=> create table nh001.store(employee_key int primary key, employee_name varchar(15), counter_no int,
age int);
```
A local temporary table "data_to_delete" is also created with employee id.
```
dbadmin=> CREATE LOCAL TEMP TABLE data_to_delete (emp_id INT);
```
Load the "employee_to_delete.txt" file , which has multiple delete statements to "data_to_delete" table.
```
dbadmin=> COPY data_to_delete FROM '/tmp/employee_to_delete.txt' ;
```

```
Rows Loaded
-------------
        15
```

Bulk Delete of rows from "shop" table

```
dbadmin=> DELETE /*+ direct */ FROM nh001.shop WHERE employee_key IN (SELECT * FROM
data_to_delete );
OUTPUT
--------
  1740
(1 row)
```

dbadmin=> DROP TABLE data_to_delete ;

To check the delete vectors and the deleted rows, use the following query. This query filters the data to the ROS containers in the initiator nodes. By querying the local data, you avoid putting load on the system tables.

```
=> SELECT schema_name, projection_name, count(*) num_ros, sum(total_row_count) num_rows,
,sum(deleted_row_count) num_deld_rows , sum(delete_vector_count) Num_dv
FROM storage_containers
WHERE node_name = ( SELECT local_node_name())
GROUP BY 1, 2
HAVING sum(deleted_row_count) > 0
ORDER BY 5 DESC;
```

Storage_containers Projection

```
schema_name | projection_name| num_ros | num_rows | num_deld_rows | Num_dv
-------------------+---------------------+------------+---------------+---------------------+-----------
nh001           | shop_b1           | 60          | 200044       |  5636                  | 62
nh001           | shop_b0           | 60          | 200210       |  5618                  | 62
```

Replay delete

Replay delete is the process of rebuilding delete vectors to adapt to the movement of records across storage containers. The replay delete operation occurs during the Tuple Mover mergeout operation. When the mergeout operation merges ROS containers that have deleted records, it purges any records that were deleted prior to the AHM epoch. Any records that cannot be purged have a new position in the newly created ROS container. This process requires rebuilding delete vectors that point to the positions of deleted records that were not purged in the new ROS container. Only deleted records that cannot be purged participate in replay delete[5].

V. DELETION OF DATA PERMANENTLY

Purge

In HP Vertica, delete operations do not remove rows from physical storage. The DELETE command in HP Vertica marks rows as deleted. Purge is the process of removing the deleted data from disk. Purge Permanently removes deleted data from physical storage so that the disk space can be reused. We can purge historical data up to and including the epoch in which the Ancient History Mark is contained. The mergeout operation merges ROS containers that qualify for mergeout. During mergeout, the Tuple Mover purges any records from qualifying ROS containers deleted prior to the AHM epoch. So the data marked for deletion will be deleted permanently from disk.

Manually purge deleted data

SELECT MAKE_AHM_NOW();
Moving AHM to last good epoch manually so that data with epoch prior to AHM gets deleted
SELECT MAKE_AHM_NOW();
Moving AHM to last good epoch manually so that data with epoch prior to AHM gets deleted.

Mergeout Delete Vectors

If there are multiple delete vectors but the percentage of deleted rows is less compared to the total rows of the table, merge the delete vectors into one delete vector. Merging the delete vectors is better than purging the entire table. Purging a table rewrites the data set of ROS containers without the deleted rows. If the number of deleted rows is small in comparison with the total rows of the table, do not purge the table. To avoid a ROS pushback, reduce the number of delete vectors.

To avoid performance degradation and ROS pushback, mergeout consolidates multiple ROS containers and purges the deleted records. The Tuple Mover performs an automatic mergeout by combining two or more ROS containers into a single container without the deleted rows. However, if the number of delete vectors per ROS containers is less than PurgeMergeoutPercent (the default is 20%), a mergeout operation does not purge the deleted records. Using many DELETE statements to delete multiple rows creates many small containers to hold the deletion marks. Each container consumes resources and impacts performance. If there are mergeout cycles for merging, the Tuple Mover merges the deletion marker containers into a single large container[9].

To merge the deleted vectors into one delete vector, use the DO_TM_TASK as follows:

```
=> SELECT DO_TM_TASK('dvmergeout');
```

Truncate

Removes all storage associated with a table, while leaving the table definition intact. TRUNCATE TABLE auto-commits the current transaction after statement execution and cannot be rolled back. TRUNCATE TABLE removes all table history preceding the current epoch, regardless of where that data resides (WOS or ROS) or how it is segmented. TRUNCATE is faster than DELETE. TRUNCATE does not involve delete vectors. Truncate acts on complete data file & not on some records. No delete vector is created during TRUNCATE operation. It removes complete data file from all projections. Its recommended, if you want to delete complete data file of a table[5].

Syntax

TRUNCATE TABLE [[*db-name.*]*schema.*]*table*

VI. CONCLUSION

Since, vertica is a data warehousing database, data are loaded in massively for performing ad-hoc queries to analyze that data stored and discover the required information for decision making process. For this, different types of COPY operations can be used according to one's requirement. And for analytical applications, read operation is executed several times when compared to write operation for which column-oriented databases like Vertica provides faster answers, because it reads only the column data files that are mentioned in the queries[10]. Also, Vertica allows the data that are deleted, to be stored in ROS container till AHM; which helps in analysis of some previous history data effectively. In this paper, simple examples are shown to make one understand how data moment happens from WOS to ROS on the execution of basic SQL operations.

REFERENCES

1. *A. Lamb, et al., "The Vertica Analytic Database : C-Store 7 Years Later", In The Proceedings of the VLDB Endowment vol.5, No.12, pp 1790–1801, 2012.*
2. *J.M.Jose, "A Detailed Study On Features of Data Warehousing Database-Vertica", International Journal of Computer Sciences and Engineering, Vol.6, No.5,pp 336-348.*
3. *C. Bear, S. Lawande, .N. Tran, B. Vandiver, S. Walkauskas, "Analytic Database Design Choices: Vertica's Experience and Perspectives", 7th Biennial Conference on Innovative Data Systems Research (CIDR '15), January 2015*
4. *Gheorghe MATEI, "Column-Oriented Databases, an Alternative for Analytical Environment", Database Systems Journal vol.1, no. 2, 2010*
5. *"Vertica Analytics Platform", Vertica Documentation, Version: 8.1.x, 2018.*
6. *H. Kyurkchiev1, K. Kaloyanova, "Performance Study of Analytical Queries of Oracle and Vertica", In the proceedings of Information Systems & Grid Technologies, March 2013.*

7. *"A DBMS Architecture Optimized for Next-Generation Data Warehousing" , The Vertica Analytic Database Technical Overview White Paper, Vertica System, 2010.*
8. *Samchu Li, "Vertica In Depth Basic introduction", May 2014*
9. *"Best Practices for Deleting Data", myVertica Vertica user community, Micro Focus, August 2017.*
10. *Bc. Martin Zboˇril, "Analysis of Vertica Database Designer", Master's Thesis report in Masaryk University Faculty Of Informatics, 2017*